**Практическая работа №5**

**Архитектура «клиент -сервер»**

Цель работы: Изучить основные принципы взаимодействия приложений, разработанных в архитектуре «клиент-сервер»,

**Теоретические сведения**

Представим, что существует машина А и на ней выполняется приложение А и существует машина В. Любая программа, которой потребуется печать на "чужом" принтере, должна включать в себя функции, подобные тем, которые выполняет приложение А. Но нагружать этими стандартными действиями каждое приложение — текстовые и графические редакторы, системы управления базами данных и другие приложения — не очень рационально. Гораздо выгоднее создать специальный программный модуль, который будет выполнять формирование сообщений-запросов к удаленной машине и прием результатов для всех приложений. Такой служебный модуль называется клиентом.

На стороне же компьютера В должна работать другая специализированная программа — сервер, постоянно ожидающий прихода запросов на удаленный доступ к ресурсу (принтеру или файлам, расположенным на диске) этого компьютера. Сервер, приняв запрос из сети, обращается к локальному периферийному устройству с участием локальной ОС.

На сегодняшний день появилась возможность иметь вычислительные и информационные ресурсы на рабочем столе пользователя и управлять ими по собственному желанию с помощью графического интерфейса. Увеличение производительности персональных компьютеров позволило перенести части системы (интерфейс с пользователем, прикладную логику) для выполнения непосредственно на рабочем месте, а функции обработки данных оставить на центральном компьютере. Таким образом, система стала распределенной - одна часть функций выполняется на центральном компьютере, другая - на персональном, который связан с центральным посредством коммуникационной сети. Появилась клиент-серверная модель взаимодействия компьютеров и программ в сети и на этой основе стали развиваться средства разработки приложений для реализации информационных систем.

Очень удобной и полезной функцией клиентской программы является способность отличить запрос к удаленному файлу от запроса к локальному файлу. Если клиентская программа умеет это делать, она сама распознает и перенаправляет (redirect) запрос к удаленной машине. Отсюда и название, часто используемое для клиентской части — редиректор. Иногда функции распознавания выделяются в особый программный модуль, в этом случае редиректором называют не всю клиентскую часть, а только этот модуль.

Программные клиент и сервер выполняют системные функции по обслуживанию запросов всех приложений компьютера А на удаленный доступ к файлам компьютера В. Чтобы приложения компьютера В могли пользоваться файлами компьютера А, описанную схему нужно симметрично дополнить клиентом для компьютера В и сервером для компьютера А. Схема взаимодействия клиента и сервера с приложениями и локальной операционной системой приведена на рисунок 5.1.

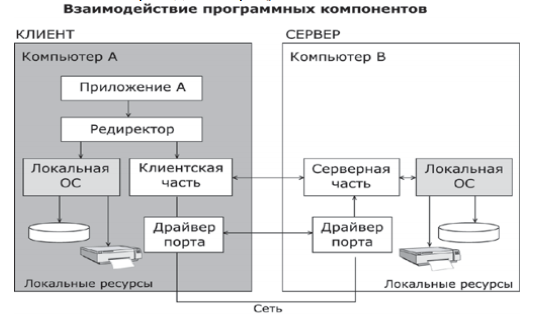


Рисунок 5.1 - взаимодействие клиента и сервера

В последнее время долю обработки, приходящуюся на клиента, стали называть "толщиной" клиента. Развитие архитектуры "клиент-сервер" происходит по спирали и в настоящее время намечается тенденция централизации вычислений, то есть замены "толстых" клиентов - рабочих станций на основе высокопроизводительных ПЭВМ, оснащенных мощным программным обеспечением для поддержки прикладных программ, мультимедийных средств, навигационного и графического интерфейса - "тонкими" клиентами. Характерный пример "тонкого" клиента - архитектура Sun Ray Hot Desk, предложенная компанией Sun Microsystems.

Архитектура Sun Ray Hot Desk предполагает использование настольных систем, имеющих минимум программных и аппаратных средств, но обладающих широкими возможностями работы с приложениями в соответствии с основной идеей "тонких" клиентов - вынести на сервер все, вплоть до виртуальных драйверов устройств, включая драйвер монитора (рисунок 5.2).

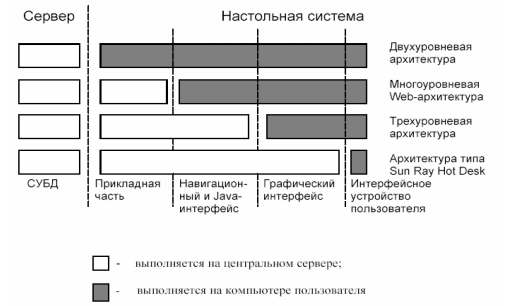


Рисунок 5.2 - Ранжирование клиентов по "толщине"

Компоненты "клиент" и "сервер" не обязательно должны работать на разных машинах, хотя обычно это так и есть – клиентское приложение находится на рабочей станции пользователя, а сервер - на специальной выделенной машине. Наиболее распространены следующие виды серверов: файл-серверы, серверы баз данных, серверы печати, серверы электронной почты, Web-сервер и другие. Однако двухуровневая архитектура "клиент-сервер" имеет также недостатки, такие, например, как сложность администрирования и низкая информационная безопасность.

Что можно отнести к категории клиент-сервер, а что нет, до сих пор является предметом жарких дискуссий. Также как и вопрос о том, что можно считать объектно-ориентированным, а что - нет. В нашем случае будет достаточно определения решений на базе клиент-сервер как децентрализованной архитектуры, позволяющей конечным пользователям получать гарантированный доступ к информации в разнородной аппаратной и программной среде. Приложения клиент-сервер сочетают пользовательский графический интерфейс клиента с реляционной базой данных, расположенной на сервере. Структура таких приложений подразумевает возможность совместной работы пользователей; при этом ответственность за выполнение тех или иных функций ложится на различные, независимые друг от друга элементы открытой распределенной среды. Берсон далее утверждает, что приложение клиент-сервер обычно можно разделить на четыре компонента:

• Логика представления. Часть приложения, обеспечивающая связь с инструментами конечного пользователя. Таким инструментом может быть терминал, считыватель штрих-кодов или переносной компьютер. Включает функции: формирование изображения, ввод и вывод информации, управление окнами, поддержка клавиатуры и мыши.

• Бизнес-логика. Часть приложения, использующая информацию, вводимую пользователем, и информацию, содержащуюся в базе данных, для выполнения транзакций, удовлетворяющих бизнес-правилам.

• Логика базы данных. Часть приложения, &quot;манипулирующая данными приложения. В реляционной базе данных подобные действия обеспечиваются с помощью языка SQL (SQL, Structured Query Language, язык структурированных запросов).

• Механизмы обращения. Непосредственная работа с базой данных, к базе данных выполняемая СУБД. В идеальном случае механизмы СУБД прозрачны для бизнес-логики приложения.

# Разработка клиент-серверной системы с примерами UML-диаграмм

Сеть - это такая конфигурация вычислительных машин, в которой они обмениваются информацией между собой.

Для того чтобы сеть функционировала должным образом, информация, сформированная отправителем, должна быть передана по линии коммуникации и доставлена определенному получателю в понятной форме.

В связи с тем, что для выполнения этих функций требуется взаимодействие различных сетевых программных и технических средств, разработчики сети создали концепцию семейства (или комплекта) коммуникационных протоколов.

Сетевой протокол представляет собой набор формальных правил, которые объясняют каким образом программное обеспечение и технические средства должны взаимодействовать в сети для передачи информации.

Семейство протоколов Internet составляет одну группу из таких сетевых протоколов. Оно группируется вокруг протокола Internet (IP). Другими членами семейства протоколов Internet являются:

* протокол управления передачей (TCP);
* пользовательский датаграмный протокол (UDP);
* протокол определения адреса (ARP);
* протокол определения обратного адреса (RARP);
* протокол управляющих сообщений Internet (ICMP).

TCP/IP обеспечивает сервис для многих различных типов узловых компьютеров, присоединенных к гетерогенным (разнородным) сетям.

TCP/IP первоначально был разработан министерством обороны США для ARPANET, региональной сети с коммутацией пакетов, и впервые продемонстрирован в 1972 г.

НТТР - сервер - программа, которая работает на основе протокола HTTP (протокол передачи гипертекста). Данная программа обслуживает запросы, поступающие от НТТР – клиентов., который является протоколом прикладного уровня для распределенных, гипермедиа систем информации. Это - общий, не имеющий гражданства, объектно-ориентированный протокол, который может использоваться для многих задач. Характеристика HTTP - занесение и согласование, предоставление данных.

HTTP также используется в качестве общего протокола для связи между агентами пользователя и серверами/шлюзами в других системах Internet, включая те, которые поддерживают SMTP, NNTP, FTP, Gopher и протоколы WAIS. Таким образом, HTTP допускает гипермедиа доступ к ресурсам, доступным из разнообразных приложений.

Основной задачей HTTP-сервера является обеспечение надежного взаимодействия между клиентом (приложением клиента) и сервером, предоставляющим информационный ресурс. Для реализации взаимодействия "клиент-сервер" важно, какой метод HTTP запроса использует клиентская часть при обращении к WWW серверу. В общем случае, запрос - это сообщение, посылаемое клиентом серверу. Первая строка HTTP запроса включает в себя метод, который должен быть применен к запрашиваемому ресурсу, идентификатор ресурса (URI - Uniform Resource Identifier), и используемую версию HTTP-протокола. Существует несколько методов, но основными являются GET и POST. Метод POST используется для запроса серверу, чтобы тот принял информацию, включенную в запрос, как относящуюся к ресурсу, указанному идентификатором ресурса. Метод GET используется для получения любой информации, идентифицированной идентификатором ресурса в HTTP запросе.

Рассмотрим подробнее протокол HTTP и функции сервера. HTTP- протокол является протоколом запроса/ответа. HTTP-соединение инициируется пользователем и состоит из запроса к ресурсу определенного сервера. В простейшем случае, соединение представляет собой поток данных между клиентом — инициатором соединения и сервером:

IMG_256

Pисунок 5.3 -Соединение между клиентом и сервером

Клиент посылает запрос в сервер в форме метода запроса, URI и версия протокола, сопровождающийся MIME-подобным сообщением, содержащим модификаторы запроса, информацию клиента и возможно содержание тела. Сервер указывает строку статуса, включая версию протокола сообщения и успех или код ошибки, сопровождающиеся MIME-подобным сообщением, содержащему информацию сервера, метаинформацию ответа и возможно содержание тела ответа. Более подробно это описано в RFC 2068.

HTTP-связь обычно происходит над TCP/IP соединениями. Встроенный порт - TCP 80, но другие порты также могут быть использованы. Это не означает что HTTP может быть реализован на основе только TCP, но также он может быть реализован и в других сетях, не обязательно только в Internet. Это означает, что HTTP предполагает использование надежного транспорта и любой протокол, который удовлетворяет этим требованиям, может быть использован.

До устойчивых соединений устанавливалось отдельное соединение TCP, чтобы выбрать каждый URL, повышая загруженность HTTP серверов и Internet. Использование встроенных образов и других связанных данных часто требует от клиента делать многочисленные запросы к тому же самому серверу за короткий промежуток времени. Поэтому перешли на устойчивые соединения. Устойчивые HTTP-соединения имеют многие преимущества:

* открывая и закрывая меньшее количество соединений TCP, сохраняется машинное время и сокращается размер память, используемой для управления блоками TCP;
* HTTP-запросы и ответы могут обрабатываться конвейерным методом.
* Конвейер позволяет клиенту делать многочисленные запросы не ожидая каждого ответа, используя одно TCP соединение, за счет чего экономиться уйма времени.
* Загруженность сети уменьшается за счет уменьшения количества TCP пакетов, организующих соединение;
* HTTP может развиваться более грациозно, поскольку ошибки могут быть доведены до клиента без разрыва соединения.

Очень важно чтобы сервер правильно обрабатывал заголовок запроса.

Все выше написанное касается протокола HTTP 1/1, а для протокола HTTP 1/0 характерно следующее: сервер НЕ ДОЛЖЕН устанавливать устойчивое соединение с клиентом HTTP/1.0.

При использовании устойчивых соединений нет ограничений на интервал времени, в течении которого сервер будет ожидать данных от клиента, а по истечении которого разрывал бы соединение и посылал бы сообщение об ошибке.

Клиент или сервер МОГУТ закрыть соединение в любое время.

Например, клиент начинал посылать новую просьбу, а в это время сервер решил закрыть соединение (прекратить "ожидание"). С точки зрения сервера, соединение закрывается по причине ошибки, но с точки зрения клиента – все нормально и он хочет работать дальше. Это означает, что клиенты, серверы и серверы ДОЛЖНЫ быть способными восстанавливать соединение.

Программное обеспечение клиента ДОЛЖНО вновь открывать соединение и повторно передавать запрос при использовании метода GET; другие методы НЕ ДОЛЖНЫ автоматически были делать новую попытку соединения, хотя в конечном счете все решает оператор. Тем не менее, эта автоматическая повторная попытка НЕ ДОЛЖНА повторяться, если вторая просьба была неудачной.

Серверы ДОЛЖНЫ всегда отвечать по крайней мере на один запрос на соединение, если это возможно. Серверы НЕ ДОЛЖНЫ закрывать соединение в середине передачи ответа.

Клиенты, которые используют устойчивые соединения, ДОЛЖНЫ ограничивать количество одновременных соединений, которые они поддерживают в данном сервере. Клиент единственного потребителя ДОЛЖЕН поддержать САМОЕ БОЛЬШЕЕ 2 соединения с любым сервером. Эти руководящие принципы собираются улучшить время ответа HTTP и избегать перегрузки Internet или других сетей.

Так как для реализации программы будет использован протокол транспортного уровня, далее разговор пойдет о механизме сокетов. Сокеты предоставляют независимые от протокола услуги по организации сетевого интерфейса и способны обеспечить работоспособность методов IPC (определенных стандартом POSIX) в масштабах локальной сети. В частности, сокеты позволяют работать с протоколом TCP, что нам и нужно. Обращаться к сокету можно по IP-адресу хост-машины и номеру порта. Заданный таким образом адрес уникален в масштабах всей Internet, так как для каждой машины комбинация адреса и номера порта уникальна.

Различают гнезда с установлением соединения (т.е. адреса сокетов отправителя и получателя выясняются заранее, до передачи сообщений между ними) и без установления соединения (адреса гнезд отправителя и получателя передаются с каждым сообщением, посылаемым из одного процесса в другой). При использовании сокетов обычно применяются следующие стандартные домены: AF\_UNIX (формат адреса – путевое имя UNIX) и AF\_INET (формат адреса – хост-имя и номер порта). Вот второй-то нам как раз и нужен. Протокол TCP создает виртуальный канал, т.е. используются сокеты с установлением соединения, что означает гарантированность доставки сообщения, а также их очередность прихода идентичную очередности отправления.

Интерфейсы прикладного программирования сокетов перечислены ниже: Socket – создает сокет заданного типа и с указанным протоколом для конкретного домена.

Bind – присваивает сокету имя.  
Listen – задает количество ожидающих клиентских сообщений, которые можно поставить в очередь к одному серверному сокету.  
Accept – принимает запрос на соединение от клиентского сокета.  
Connect – посылает запрос на соединение в серверный сокет.  
Send – передает сообщение в удаленный сокет.  
Recv – принимает сообщение из удаленного сокета.  
Close – закрывает сокет.

## Пример диаграмм "USE Case"

Системы не существуют в изоляции. Как правило, они взаимодействуют с актерами - людьми или программами - которые используют систему в своих целях, причем каждый актер ожидает, что она будет вести себя определенным, вполне предсказуемым образом. Прецедент (Use case) специфицирует поведение системы или ее части и представляет собой описание множества последовательностей действий (включая варианты), выполняемых системой для того, чтобы актер мог получить определенный результат.

С помощью прецедентов можно описать поведение разрабатываемой системы, не определяя ее реализацию. Таким образом, они позволяют достичь взаимопонимания между разработчиками, экспертами и конечными пользователями продукта. Кроме того, прецеденты помогают проверить архитектуру системы в процессе ее разработки. Реализуются они кооперациями.

Хорошо структурированные прецеденты описывают только существенное поведение системы или подсистемы и не являются ни слишком общими, ни слишком специфическими.

Ниже приведен пример использования диаграмм «Use Case».

На рисунке 5.4 показано поведение системы со стороны клиентской части системы. А на рисунке 5.5 показано поведение системы со стороны серверной части.

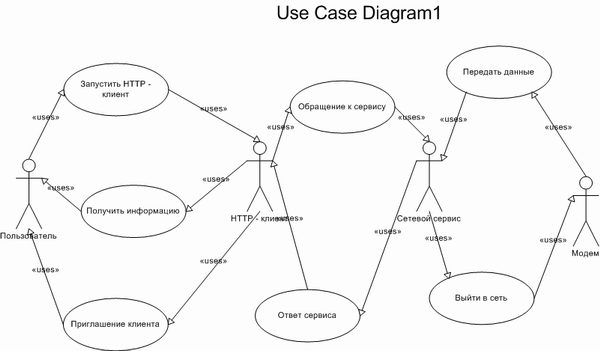


Рисунок 5.4 - Поведение клиентской стороны

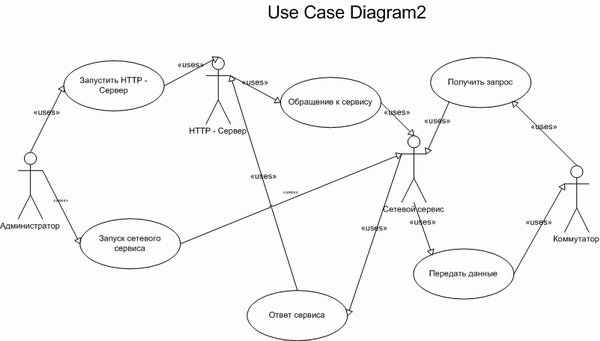


Рисунок 5.5 -

## Пример диаграмм "Deployment"

Диаграммы развертывания, или применения, - это один из двух видов диаграмм, используемых при моделировании физических аспектов объектно- ориентированной системы. Такая диаграмма показывает конфигурацию узлов, где производится обработка информации, и то, какие компоненты размещены на каждом узле.

Диаграммы развертывания используются для моделирования статического вида системы с точки зрения развертывания. В основном под этим понимается моделирование топологии аппаратных средств, на которых выполняется система. По существу, диаграммы развёртывания - это просто диаграммы классов, сосредоточенные на системных узлах.

Диаграммы развертывания важны не только для визуализации, специфицирования и документирования встроенных, клиент-серверных и распределенных систем, но и для управления исполняемыми системами с использованием прямого и обратного проектирования.

Ниже приведен пример использования диаграмм «Deployment».

На рисунке 5.6 смоделирована клиент-серверная система. А на рисунке 5.7 показана полностью распределенная система.

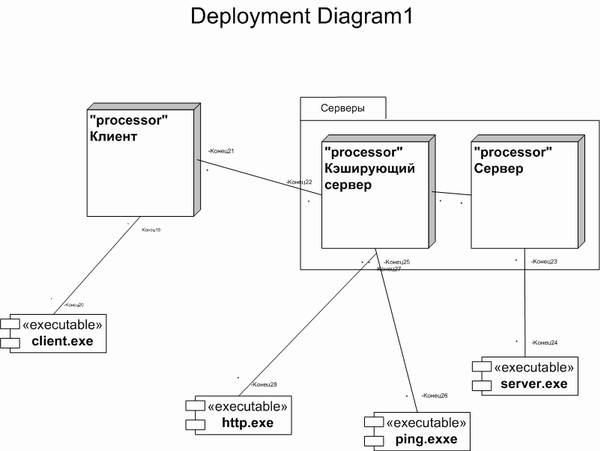


Рисунок 5.6 - Моделирование клиент-серверной системы

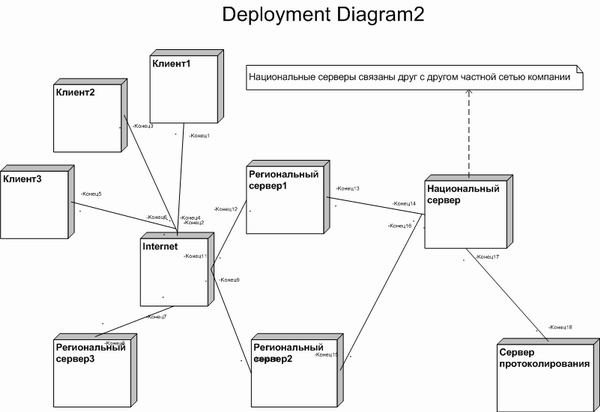


Рисунок 5.7 - Моделирование полностью распределенной системы

**Задание на практическую работу**

Используя диаграммы прецедентов и диаграммы развертывания, необходимо смоделировать работу системы клиент- серверного приложения для отправки сообщений электронной почты с мобильного телефона. Стоит учесть, что помимо этого, сервис имеет ещё веб-сервис, в котором также имеется возможность отправки сообщений. Мобильное приложение имеет возможность интеграции нескольких почтовых сервисов.